Effektiv integrasjon av Java servlets i Apache web-tjener

Hovedoppgave
ved
sivilingeniørutdanningen i
informasjons- og kommunikasjonsteknologi

av
Steffen S. Hellestøl

Grimstad, mai 2000

Forord

Hovedoppgaven "Effektiv integrasjon av Java servlets i Apache web-tjener", er skrevet i forbindelse med det avsluttende diplomarbeidet ved sivilingeniør-utdanningen for informasjon- og kommunikasjonsteknologi ved Høgskolen i Agder, Grimstad.

Oppgaven ble gjennomført i vårsemesteret, januar til og med mai 2000.
Jeg ønsker spesielt å takke Øyvind Hanssen for en interessant oppgave
og også mine øvrige veiledere Jan P. Nytun og Mikael Snaprud.

Grimstad, 2. juni 2000
Steffen S. Hellestøl

Sammendrag

I hovedoppgaven "Effektiv integrasjon av Java servlets i Apache web-tjener" belyses design og implementasjon av serverapplikasjoner som er skrevet i språk som C men hvor funksjonalitet kan utvides med dynamisk pluggbare komponenter skrevet i Java. Oppgaven fokuserer spesielt på fordeler og ulemper ved en arkitektur hvor Java virtuell maskin (og java komponentene) kjører innenfor samme prosess som tjeneren, i forhold til en annen arkitektur hvor tjeneren og JVM kjører i separate prosesser og kommuniserer ved hjelp av operativsystemets IPC mekanismer (TCP sockets).

I oppgaven er det gitt eksempler på design som integrerer Java servlets med webtjeneren Apache. Det er lagt spesiell fokus på to design. Det ene er i utstrakt bruk ute på Internett i dag i en løsning kalt Apache JServ. Det andre designet bygger på en idé om hvordan det kan være mulig å forbedre JServ ved å bytte ut IPC mekanismen med en protokoll som kommuniserer med JVM ved hjelp av Java Native Interface.

Den viktigste oppdagelsen som er gjort i denne oppgaven er at det alternative designet som har vært undersøkt ikke lett lar seg gjennomføre på grunn av måten Apache v1.3 arbeider med prosesser og tråder. Dette vil imidlertid trolig endre seg ved lanseringen av Apache v2.0 siden denne da får en mer fleksibel prosess- og trådmodell.

Innholdsfortegnelse

1. INNLEDNING 1

1.1 Mål 1

1.2 Avgrensning 2

1.3 Rapportens oppbygning 2

2. METODE 3

3. GRUNNLEGENDE TEKNOLOGIER 4

3.1 World Wide Web 4

3.1.1 Historikk 5

3.1.2 Arkitektur 5

3.1.3 HTML, HyperText Markup Language 6

3.1.4 HTTP, HyperText Transfer Protocol 6

3.2 Apache 8

3.2.1 Historikk 8

3.2.2 Arkitektur 9

3.2.3 Konfigurasjon 11

3.2.4 Modul API 13

3.2.5 Apache 2 15

3.2.6 Dynamisk innhold 17

3.3 Java 18

3.3.1 Java Virtual Machine, JVM 18

3.3.2 Java Native Interface, JNI 19

3.3.3 Java Servlets 20

3.4 Oppsummering 21

4. JAVA SERVLET ARKITEKTURER 23

4.1 Servlet i selvstendig prosess (JServ) 23
4.1.1 Designskisse 24
4.1.2 Fordeler og ulemper 25
4.1.3 Bruk av JNI for å effektivisere kommunikasjon 28

4.2 Servlet i webserver prosess (JNIServ) 28
4.2.1 Designskisse 29
4.2.2 Fordeler og ulemper 29

4.3 Servlet i JavaWebServer 34
4.3.1 Designskisse 34
4.3.2 Fordeler og ulemper 35

4.4 Java men ikke servlet 36
4.4.1 Designskisse 37

4.5 Kvalitativ sammenligning 38
4.5.1 Skalerbarhet 38
4.5.2 Effektivitet og transaksjonskostnader 38
4.5.3 Sikkerhet 39
4.5.4 Stabilitet og feiltoleranse 39
4.5.5 Kompleksitet og brukervennlighet 39
4.5.6 Portabilitet 40
4.5.7 Helhetsvurdering og konklusjon 40

4.6 Kvantitativ sammenligning av ytelse 40

4.7 Oppsummering 41

5. KONKLUSJON 42

LITTERATURLISTE 43

Figurer

Figur 3.1 HTTP protokollstakk....7
Figur 4.1 Apache JServ arkitektur....24
Figur 4.2 Webserver og JVM i samme prosess....29
Figur 4.3 Java basert webserver med servlets....34
Figur 4.4 Java baserte Apache moduler....37

Tabeller

Tabell 3.1 Metoder som kan brukes i HTTP spørringer....7
Tabell 3.2 Viktige containere som begrenser et direktivs kontekst....13
Tabell 3.3 Livssyklus til en spørringer i Apache....14
Tabell 3.4 Moduler i Apache som genererer dynamiske websider....17
Tabell 3.5 Terminologi brukt i servlet rammeverk....20
Tabell 4.1 Klassifisering av arkitekturer....41

Vedlegg

Vedlegg 1 Oppgaveforslag, Øyvind Hanssen
Vedlegg 2 Oppgave teksten, Steffen S. Hellestøl

1. Innledning

Denne oppgaven belyser design og implementasjon av tjenerapplikasjoner som er implementert i språk som C men hvor funksjonalitet kan utvides med dynamisk pluggbare komponenter skrevet i Java. Det er blant annet gjort en undersøkelse av fordeler og ulemper ved en arkitektur hvor Java virtuell maskin (og java komponentene) kjører innenfor samme prosess som tjeneren, i forhold til en arkitektur hvor tjeneren og JVM kjører i separate prosesser og kommuniserer ved hjelp av operativsystemets IPC mekanismer (TCP sockets). Mer konkret handler oppgaven om Java Servlets som en utvidelse av web-tjenere. Java servlets er tjener-motstykket til Applets. Servlets er Java komponenter som kjører på tjener-siden. En servlet instans er i stand til å motta HTTP anrop og generere et web-innhold (HTML) som svar.

Det finnes ulike alternativer til hvordan servlets kan integreres med (eksisterende) web-tjenere:

- Hele web-tjeneren er implementert i Java (og kan lett utvides med servlets).
- Web-tjener og Java virtuell maskin (med servlets) kjører i separate prosesser. Disse bruker IPC (TCP-socket) og en egen protokoll for å kommunisere seg i mellom. Det er dette som er tilfelle med Apache JServ (servlet støtte for den populære Apache web-tjener).
- Et tredje alternativ er å inkludere JVM i samme prosess som web-tjener. Apache prosjektet har også jobbet med dette. Det vil si de har vurdert et modul for Apache som inkluderer JVM, men dette arbeidet ser ut til å ha stoppet opp.

1.1 Mål

Målsetningen for oppgaven har vært å:

1. Gi en oversikt over forskjellige måter en Java servlet kan implementeres i web-tjeneren Apache.

2. Gjøre en sammenlikning av disse og drøfte fordeler og ulemper ved hvert av løsningsalternativene med hensyn til begrensninger, robusthet, sikkerhet, skalerbarhet, ytelse osv.
3. Implementere en Java servlet arkitekturen hvor JVM og Apache kjører i samme prosess og kommuniserer ved hjelp av Java Native Interface (JNI).
4. Gjøre ytelsesmålinger av prototypen for å finne ut hvordan denne kvantitativt kan sammenlignes med de andre løsningene.

1.2 Avgrensning

Ved prosjektstart var det planlagt at det skulle gjøres en implementasjon av en prototyp slik at det uttrykt i punkt 3 og 4 i forrige avsnitt. Teknologistudien var fra begynnelsen siktet mot at en prototyp skulle lages, men etter en vurdering av de kvalitative egenskapene til de forskjellige elementene som inngår i denne løsningen ble det imidlertid funnet ut at denne arkitekturen på det nåværende tidspunkt ikke var mulig å gjennomføre.

1.3 Rapportens oppbygning

Rapporten er delt i to hoveddeler. Den første delen gir en innføring i de viktigste teknologiene; World Wide Web, webtjener arkitektur med case studie av Apache, og Java.
I den andre delen presenteres de forskjellige arkitekturene jeg har funnet frem til. Det gjøres en vurdering av hvert av disse og kapittelet avsluttes med en konklusjon.

2. Metode

I arbeidet med hovedoppgaven skulle det vurderes alternative arkitekturer til Apache JServ. Vurderingene som er gjort er utelukkende basert på kvalitative undersøkelser; for eksempel med hensyn til skalerbarhet, effektivitet og feiltoleranse. Som grunnlag for disse vurderingene har jeg gjort en forstudie av de teknologier som ligger til grunn for de alternative arkitektuerne. Bakgrunnstoffet i studien er hentet fra bøker og websider (se referanser bak i rapporten). Jeg har også hentet idéer fra diskusjonsgrupper ute på nettet og sammen med mine egne vurderinger prøvet å sammenstille dette til et hele. Jeg har ikke gjort noen form for kvantitativ undersøkelse av ytelse.

3. Grunnlegende teknologier

Denne delen av rapporten gir en innføring i viktige begreper om teknologi som det forutsettes at leseren er kjent med før han går videre til neste kapittel. Avsnitt 3.1 handler om World Wide Web, litt om webbens historie og om protokollene som ligger bak denne tjenesten. Avsnitt 3.2 er den viktigste og tyngste delen av kapittelet og er viet web-tjeneren, "Apache". Fokus for dette avsnittet er serverens oppbygning og modul arkitektur. Avsnitt 3.3 handler om Java; Java Virtual Machine (JVM), Java Native Interface (JNI) og Java servlets. Til slutt avrundes kapittelet med en oppsummering.

3.1 World Wide Web

World Wide Web eller "webben" har fått mye av æren for den utbredelsen og populariteten Internett i dag har. Måten vi tenker om- og måten mange av oss behandler informasjon har endret seg. Vi står i dag foran et paradigmeskifte hvor de klassiske programvareprodusentene endrer profil fra å levere software i pappesker til å bli tjeneste leverandører, populært kalt; "ASP'er" eller "application service providers". I fremtiden vil mange av applikasjonene vi i dag ser i eskene bli levert over nettet. I tillegg vil vi få et utall nye tjenester som vil kunne tilby et nytt nivå av interaktivitet gjennom rik sanntidskommunikasjon. Vi ser blant annet endringen gjennom den konvergensen som nå skjer mellom IT- og mediaselskapene. Og den viktigste muliggjørende teknologien oppi alt dette er nettopp webben.

3.1.1 Historikk

Webben har sitt utspring i forskningsmiljøet ved CERN [23] hvor det hadde oppstått et behov for en effektiv måte å distribuere elektroniske dokumenter. Som et resultat av dette utviklet man i 1990 en protokoll, HyperText Transfer Protokoll (HTTP), som standardiserer kommunikasjon mellom server og klient. Et eget dokumentformat, HyperText Markup Language (HTML), ble brukt til å formatere teksten og til å knytte sammen dokumenter ved hjelp av såkalte hyperlenker. Ved hjelp av en tekstbasert nettleser kunne brukerne hente frem dokumenter, og ved å trykke på ord i teksten kunne man manøvrere seg frem til andre dokumenter med mer informasjonen om det ordet man hadde valgt. I 1993 slapp National Center for Supercomputing Applications (NCSA) ved University of Illinois en nettleser med et grafisk grensesnitt. Nettleseren fikk navnet Mosaic og kunne vise bilder i teksten og man kunne navigere ved hjelp av mus. På midten av 90-tallet hadde webben fått millioner av brukere. Mye av grunnen til at webben fikk et så stort gjennomslag på så kort var fordi protokollene var basert på åpene standarder, uten lisensbegrensninger. Det gjorde det mulig å lage gratis versjoner av nettlesere og web servere.

3.1.2 Arkitektur

I bunnen av webben ligger en klient-tjener arkitektur. Klienten, eller nettleseren, adresserer innholdet ute på nettet ved hjelp av en "Uniform Resource Locator" (URL). URL'en inneholder eksplisitt informasjon om hvordan man skal få tilgang til en ressurs. Den kan inneholde informasjon om hvilken protokoll som skal benyttes, adressen til hvor tjeneren befinner seg på nettet, hvor i tjeneren ressursen er lagret og annen informasjon som er nødvendig for å kunne få tilgang til ressursen. En URL er typisk bygget opp på følgende måte:

<protokoll>//<server>/<ressurs>

En URL som benytter en HTTP-protokoll kan for eksempel se slik ut:

http://siving.hia.no/usa

HyperText Transfer Protocol (HTTP) brukes mellom webserveren og nettleseren for å overføre datafiler. Disse datafilene kan være statiske og ressursnavnet må da mappes til en fil på serverens harddisk som så sendes tilbake til klienten. Det er mulig å overføre alle filtyper på denne måten; det kan for eksempel være hypertekst dokumenter (HTML), bilder, video el. Det er også mulig å få serveren til å generere innholdet dynamisk i det øyeblikket nettleseren gjør spørringen til serveren. Slike skreddersydde sider inneholder ofte informasjon som er hentet fra en database. Med HTTP er det også mulig å sende informasjon tilbake fra nettleseren og for eksempel lagre denne informasjonen i en database. Det viktigste filformatet som transporteres av HTTP-protokollen er HTML-dokumenter.

3.1.3 HTML, HyperText Markup Language

HyperText Markup Language (HTML) er et filformat for dokumenter som skal publiseres ute på webben. Med HTML er det mulig å tagge utvalgte deler av skriften og tilordne spesielle egenskaper. For eksempel er det mulig å spesifisere at en linje med tekst skal være en overskrift, eller at den skal fungere som en link. Et HTML-dokument kan være lagret statisk på serveren eller det kan genereres dynamisk av serveren. HTML-dokumenter kan inneholde skjema som brukeren kan fylle ut for å sende informasjon tilbake til serveren.

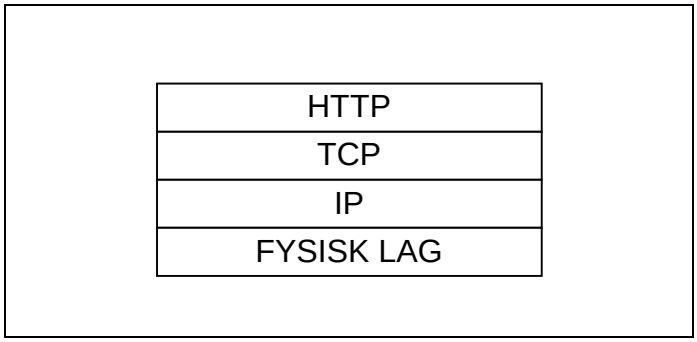
3.1.4 HTTP, HyperText Transfer Protocol

HyperText Transfer Protocol (HTTP) er den underliggende protokollen som brukes i alle webservere og nettlesere. HTTP standardiserer måten nettlesere gjør spørringer og hvordan webservere svarer på disse spørringene. HTTP versjon 1.0 er beskrevet i RFC 1945 [16], og HTTP versjon 1.1 i RFC2616 [17]. En typisk interaksjon mellom en nettleser og en webserver begynner med at nettleseren setter opp en TCP/IP socket til webserveren, vanligvis på port 80. Etter at forbindelsen er satt opp sender nettleseren en ASCII formatert spørring til serveren. HTTP versjon 1.0 definerer et sett metoder som kan brukes på serveren for å få tilbake forskjellige typer respons. Disse er oppsummert i tabell 3.1 på neste side.

Tabell 3.1 Metoder som kan brukes i HTTP spørringer

Metode	Funksjon
GET	Henter en ressurs fra serveren. Nettleseren sender en parameter sammen med GET-spørringen for å fortelle tjeneren hvilken ressurs som søkes. Tjeneren responderer med å sende et header-sett med blant annet metainformasjon om innholdet som returneres. Til slutt sendes selve innholdet.
HEAD	Henter header-settet for en ressurs. Denne metoden er ekvivalent med GET med unntak av at innholdet blir utelatt.
POST	Sender informasjon til en ressurs i serveren. Denne metoden er ekvivalent med GET, men tillater også at nettleseren sender ekstra informasjon i form nøkkel-/verdi variabler. POST brukes ofte sammen med utfylling av skjema, eller forms, i HTML dokummenter.
OPTIONS	Returnerer en liste over metoder som serveren tillater at klienten bruker.
TRACE	Returnerer et standard header-sett etterfulgt av en kopi av header settet som ble sendt av klienten.
DELETE	Sletter en ressurs på serveren.
PUT	Oppretter eller endrer en fil på serveren.
CONNECT	Brukes blant annet i forbindelse med Secure Sockets Layer (SSL).

HTTP er en forbindelsesløs protokoll. Etter at en spørring er utført og respons er mottatt fra serveren, lukkes TCP/IP forbindelsen. Ingen tilstandsinformasjon lagres mellom hver interaksjon. I web applikasjoner hvor dette er et behov må andre mekanismer tas i bruk for å kunne kjenne igjen en klient fra gang til gang, for eksempel ved hjelp av cookies.
Figur 3.1 viser lagene i protokollstakken under HTTP.



Figur 3.1 HTTP protokollstakk

3.2 Apache

Apache er navnet på en innfødt amerikansk indianer stamme, kjent for sin overlegne dyktighet i strategisk krigføring og utømmelige utholdenhet. Apache er også navnet på en fullskala web server som er utviklet gjennom et samarbeid mellom frivillige individuelle utviklere. Apache er derfor også gratis og kildekoden ligger åpent tilgjengelig for alle å se og bruke. Koordinering, planlegging og utviklingen av Apache prosjektet gjøres ved hjelp av Internett og webben.

Apache er i dag den mest brukte webserver på Internett. I følge målinger utført av Netcraft [24] i mai 2000, har Apache en andel på over 60 prosent av web-server markedet. Det kan være flere grunner til at Apache har blitt så populær. Programvaren er gratis, men kanskje viktigst er at alle har innsyn til kildekoden. Dette gjør at bugs og hull i sikkerheten oppdages og fikses hurtigere.
At kildekoden er åpen gjør det også mulig å tilpasse serveren til spesielle formål. Apache har dessuten et ekstensivt sett av direktiver som kan brukes til å konfigurere serveren. Direktivene forteller Apache hvordan spørringer fra klienter skal håndteres og generelt hvordan serveren skal fungere. Det er også mulig å legge til nye direktiver med ny funksjonalitet ved hjelp av statisk eller dynamisk linkede plugginn moduler.

3.2.1 Historikk

Apache har sitt opphav fra National Center for Supercomputing Applications (NCSA) ved University of Illinois hvor man tidlig på 90-tallet hadde laget en webserver som gikk under navnet "httpd". Denne var gratis og kildekoden var tilgjengelig for alle. I 1995 var dette den mest brukte webserveren ute på nettet, men siden en av nøkkelpersonene ved NCSA hadde sluttet, var utvikling av serveren stoppet opp. Som et resultat av dette hadde mange web administratorer laget sine egne utbedringer og feilfikser. Disse manglet imidlertid en felles

distribusjon, og noen slike administratorer formet derfor en gruppe for å samordne disse "patchene". Distribusjonen deres fikk navnet "A PAtCHy server" eller Apache. Etterhvert ble flere nye funksjoner lagt til og serveren fikk også en ny arkitektur med kodenavnet "Shambhala". Utviklingen av Apache koordineres i dag av Apache Software Foundation [7] som også driver en rekke underprosjekter med relevans til webserveren.

3.2.2 Arkitektur

Server arkitekturen til Apache går ofte under kodenavnet Shambhala. Da Apache-gruppen skrev om serveren ble det lagt vekt på at den skulle være modulert oppbygd. Man definerte en API for pluggbare moduler og samtidig fjernet man mesteparten av funksjonaliteten i kjernen og plasserte den i nye pluggbare moduler. Størrelsen på serverkjernen ble redusert og fikk kun et minimalt sett av operasjoner. Ved å plassere mye av funksjonaliteten i moduler kunne administratorer strømlinjeforme og optimalisere sine servere. I tillegg åpnet den nye server API'en for at tredjeparts utviklere lettere kunne legge til nye funksjoner på en strukturert og ryddig måte. Dette gav nye muligheter for dynamisk innhold som tidligere kun hadde vært mulig ved å bruke de mindre effektive CGI-skriptene.

I tillegg til den nye modul arkitekturen innførte man også et nytt system for prosess forgrening (forking). Hvis en server mottar flere samtidige spørringer vil responstiden bli dårligere. En klient må nødvendigvis vente til serveren er ferdig med å håndtere køen av andre spørringer som har kommet inn først, før klienten kan få behandlet sin spørring. En vanlig teknikk for å unngå dette problemet er å bruke flere prosesser som man fordeler spørringene utover. På denne måten kan flere klienter håndteres samtidig og en tung spørring som tar lang tid å behandle vil da ikke kunne fullstendig blokkere serveren fra å håndtere mindre og lettere spørringer. Gjennomsnittstiden for spørringer blir på denne måten redusert.

Et problem webadministratorer tidligere ble stilt overfor var å velge antall prosesser som skulle brukes til å håndtere spørringer. Dette kunne ofte være vanskelig å velge. Maskinen som webtjeneren kjørte på ble ofte brukt til å kjøre andre tilleggstjenester. Det var derfor viktig å ikke velge for mange prosesser fordi dette ville belaste serveren unødvendig. På den andre siden; hvis man valgte for få prosesser ville man ikke kunne oppnå den ideelle responstiden fra webtjeneren. Å sette et optimalt tall på hvor mange prosesser kunne være vanskelig fordi belastningsmønsteret på webtjeneren endret seg over tid. På enkelte tider av døgnet kunne belastningen være svært høy, mens på andre ganger, for eksempel om natten, kunne belastningen være lik null. På disse tidene kunne kapasitet i serveren brukes til andre, mer nyttige ting enn å ha et stort antall webtjener- prosesser stående på tomgang. Løsningen ble at man innførte en metode hvor prosesser startes og avsluttes dynamisk. Isteden for å sette et fast tall, fikk administratoren tre direktiver å forholde seg til; MinSpareServers, MaxSpareServers og MaxClients. I denne løsningen blir det med jevne mellomrom sjekket hvor mange serverprosesser som ikke håndterer en spørring og derfor er ledige. Hvis antall ledige tjenere er mindre enn MinSpareServers eller større en MaxSpareServers, vil Apache henholdsvis starte eller stoppe tjener prosesser. Apache har på denne måten alltid et visst antall tjener prosesser klar i bakhånd i tilfelle det kommer inn flere spørringer.
Det er også satt av et eget direktiv, MaxClients, for å sette en absolutt øvre grense for hvor mange tjener prosesser som kan tillates. Slik ungår man at en webserver som utsettes for stor trafikk uhensiktsmessig overlaster operativsystemet.

Det ligger også innbakt en sikkerhetsmekanisme i måten Apache behandler prosesser. Når man starter Apache gjøres dette som regel fra en root bruker. Httpd-prosessen starter deretter alle tjenerprosessene. Med direktivene User og Group er det mulig å bestemme hvilken bruker og brukergruppe alle tjenerprosessene skal kjøres som. Det er ofte vanlig at en bruker med begrenset tilgang til operativsystemet nyttes til dette formålet. På denne måten vil all interaksjon med omverdenen skje gjennom prosesser som ikke har full systemtilgang. Hvis uvedkommende mot formodning skulle få kontroll over

Apache, så er i det minste rettighetene til den kompromitterte prosessen høyst begrensede.

Som en siste kommentar om tjenerprosessene i Apache bør det også nevnes at disse kjøres i enkelt-trådet modus. Fordelen med denne løsningen er at dersom en av prosessene skulle krasje, så vil resten av serveren likevel fortsette å fungere. En server med én prosess og flere tråder vil ikke oppføre seg like robust og stabil. At serveren ikke bruker tråder er også en fordel hvis man skal porte Apache til plattformer uten innebygget trådstøtte. Det finne imidlertid også ulempen med denne prosess arkitekturen. Et stort problem er at serveren ikke kjører like effektivt som den kunne gjort i en tråd basert arkitektrur. Belastningen ved å kjøre mange prosesser kan være høy og kommunikasjon mellom prosessene er kostbar. Dette gjelder spesielt for operativsystem som håndterer mange prosesser dårlig. Et eksempel på dette er WindowsNT og Windows2000. I Apache versjonen for disse operativsystemene har det derfor vært nødvendig å bygge om hele serveren til heller å bruke tråder. Hele serveren kjøres da i en prosess og det som på UNIX er prosesser har i Windows versjonen blitt til tråder. I avsnitt 3.2.5 diskuteres det hvordan denne prosess- / tråd problematikken vil bli løst i neste generasjons Apache server, versjon 2.0, men først litt mer om hvordan Apache kan konfigureres.

3.2.3 Konfigurasjon

Konfigurering av Apache kan gjøres på to nivåer; før kompilering og etter kompilering.

Selv om noen foretrekker å hente ned ferdig kompilerte versjoner av Apache kan det ofte være en fordel å gjøre kompileringen selv. De viktigste valgene man må gjøre før kompilering er å bestemme hvilke moduler som skal bindes statisk inn i koden til serveren. Administratoren kan også sette standardverdier for enkelte parametre som kompileres inn i serveren. Eksempler på dette er navn på filer og kataloger som Apache bruker under kjøring. Ved å kompilere korrekte

standardverdier fra begynnelsen slipper man å oppgi disse eksplisitt når man starter httpd prosessen. Installasjonsskripet som kommer med Apache vil også skreddersy konfigurasjonsfilene slik at det kun er små endringer som behøves i ettertid.

Etter kompilering kan man tilpasse Apache på forskjellige måter gjennom et sett av direktiver. Når man starter eller eventuelt omstarter Apache, hentes disse direktivene fra konfigurasjonsfiler på harddisken. Gjennom konfigurasjonsfilene, som er kodet i vanlig ASCII, kan webadministratoren sette opp direktiver i forskjellige kontekster. Et direktiv er en kommando som får betydning for hvordan Apache håndterer spørringer. Et direktiv som settes i en konfigurasjonsfil kan alene gi mening og fungerer da som et flagg, eller det kan stå i forbindelse med et sett parametre som gis sammen med direktiver. Addhandler er et eksempel på et direktiv som sammen med to parametre kan se slik ut:

```
Addhandler geronimo .ger
```

Direktivet forteller Apache at spørringer etter filer som slutter med tegnene: ".ger" skal håndteres av en spesiell handler med navn geronimo. Hvordan en handler fungere vil bli nærmere beskrevet i neste avsnitt om Apaches modul API.

Et direktiv trenger nødvendigvis ikke få global betydning. I konfigurasjonsfilene er det mulig å definere i hvilken sammenheng, eller kontekst, direktivet skal opptre. Man kan for eksempel skrive:

```
<Location /indianere>
        Addhandler geronimo .ger
</Location>
```

Konteksten til Addhandler er her redusert av Location-containeren. Denne fører til at Addhandler kun får innflytelse på spørringer etter filer i eller under /indianere katalogen.

For eksempel vil denne adressen bli betjent av geronimo:

http://siving.hia.no/indianere/HIAvata.ger

mens denne vil ikke bli håndtert av geronimo:

http://siving.hia.no/studenter/ikt98.ger

Det er også mulig å begrense konteksten til et direktiv ved hjelp av andre typer containere. Tabell 3.2 viser et utvalg av de viktigste containerne i en Apache konfigureringsfil. Hvis et direktivet står utenfor en container får direktivet global betydning for hele serveren.

Tabell 3.2 Viktige containere som begrenser et direktivs kontekst

Container navn	Funksjon
<Limit>	Begrenser et direktiv til kun å gjelde for en spesiell HTTP-metode. For eksempel: <Limit POST>
<Directory>	Begrenser et direktiv til kun å gjelde for en spesiell kataloger på harddisken. <Directory /usr/home>
<Files>	Begrenser et direktiv til kun å gjelde for spesiell filer på harddisken. <Files *.gif>
<Location>	Begrenser et direktiv til kun å gjelde for en spesiell lokasjon på webserveren. <Location /indianere>
<VirtualHost>	Begrenser et direktiv til kun å gjelde for en virtuell server. Med dette direktivet er det mulig å sette opp flere virtuelle webservere på en maskin. De forskjellige virtuelle serverne kan adresseres ved å bruke forskjellige port, ip-adresser, og/eller HTTP anrop. <VirtualHost 128.39.202.20>

Direktivene i Apache brukes både for å konfigurere innstillinger i kjernen og i modulene. Dette betyr at settet av tilgjengelige direktiver i Apache er avhengig av hvilke moduler som er linket inn i serveren.

3.2.4 Modul API

Avsnitt 3.2.2 gav en introduksjon til Apaches "Shambhala-arkitektur". Der ble det nevnt at Apache var en modulert oppbygd server. Det betyr at all funksjonalitet som ikke er grunnleggende for at serveren skal kunne fungere er trukket ut av

kjernen og plassert i løse moduler. Disse linkes så inn etter behov. En slik modul kan enten linkes statisk eller dynamisk til serveren. En statisk linket modul må kompileres samtidig med Apache og linkes sammen med de andre objektfilene som til slutt danner den kjørbare serverfilen. Ulempen ved å bruke statisk linkede moduler er at disse alltid vil bli lastet med serveren, uavhengig om funksjonalitet i dem brukes eller ikke. Moduler som sjeldent brukes bør man derfor heller linke dynamisk. Modulen legges da i en egen fil, et dynamiske linkbart bibliotek. I UNIX kjennetegnes slike filer med benevnelsen .so og på Windows som .dll. Moduler som ligger lagret i slike bibliotek kan lastes og brukes av serveren uavhengig av når de ble kompilert. På denne måten kan webadministratoren legge til ny funksjonalitet uten å rekompilere serveren. Et `LoadModule` direktiv i konfigurasjonsfilen sørger for at modulen blir lastet og dynamisk linket til serveren under oppstart. For eksempel:

```
LoadModule mod_geronimo /usr/lib/apache/mod.geronimo.so
```

Moduler inneholder handlere som bidrar i behandlingen av en spørring. En handler er en C funksjon som er konform med Shambhala API'en. Flere handlere, ofte fra flere forskjellige moduler, har vært inne i bildet før en spørring er ferdig behandlet og transaksjonen avsluttet. For å strukturere arbeidsflyten i serveren er prosesseringen av spørringer delt inn i faser. Disse fasene er oppsummert i tabell 3.3.

Tabell 3.3 Livssyklus til en spørringer i Apache

Fase	Funksjon
1. URI til filnavn oversetting	Bestemmer hvilken fil på harddisken klienten søker tilgang til.
2. Header tolking	Les og tolk headerne fra HTTP-spørringen.
3. Aksess sjekking av vertsadresse	Sitter brukeren på en maskin med en tillatt IP adresse?
4. Autentisering av identifikasjon	Er brukeren den han utgir seg for i spørringen?
5. Autorisasjons sjekking	Har brukeren tilgang til ressursen det spørres etter?
6. Fastslå resursens MIME-type	Bestemme filtypen til ressursen det spørres etter. Dette er med å avgjøre hvilken handler som senere skal betjene spørringen.
7. Diverse	Denne fasen er sjeldent i bruk. Operasjoner som ikke passer inn i noen av de andre fasene kan legges til denne fasen.
8. Send respons tilbake til klienten	En av handlerne i Apache aksepterer å respondere på spørringen.
9. Logg transaksjonen	Skrive en kommentar til loggen om at spørringen er behandlet.

Flere moduler kan realisere handlere for en og samme fase, men det er som regel bare en av dem som aksepterer å gjøre jobben. Dette fungerer på følgende måte; Man ser på en og en modul av gangen. Hvis en modul har implementert en handler funksjonen for den aktuelle fasen vil det bli gjort et kall til denne handleren. Retur verdien avgjør det videre forløpet;

- **Avvist** - Hvis handleren avviser spørringen går jobben videre til den neste modulen som har implementert en handler for denne fasen og det gjøres et nytt forsøk der.
- **Feil** - Hvis handleren rapporterer feil avbrytes normal behandling av spørringen og som regel sendes en feilmelding tilbake til klienten.
- **Behandlet** - Hvis handleren imidlertid velger å aksepterer spørringen vil prosesseringen etterpå gå videre til neste fase. Moduler med handlere som ikke har blitt kallet før spørringen er blitt behandlet i denne fasen, vil derfor aldri få mulighet til å påvirke spørringen. Hvilke moduler som slipper først til kan endres ved hjelp av direktivene ClearModuleList og AddModule.

Øvrig konfigurering av modulene gjøres gjennom modul spesifikke direktiver. Disse tolkes og lastes inn i modulene gjennom Shambhala API'en under oppstart av serveren.

Modul API'en i Apache er basert på programmeringsspråket C. Det viktigste elementet i API'en er en dispatch tabell i en struktur som er definert med navnet "module". Det jobbes også med en lettvekts modul som skal gjøre det mulig å skrive Apache moduler i Java. Denne modulen går under navnet mod_java.

3.2.5 Apache 2

Siste utgave av Apache var versjon 1.3.12. I dette avsnittet presenters de viktigste endringene som kommer i neste generasjon, Apache 2.0, og som i øyeblikket kun foreligger i tidlige alfa utgaver. Mye av innholdet i denne delen er hentet fra en artikkel i Apache Week [25].

Den viktigste målsetningen for versjon 2 har vært å gjøre det lettere å porte Apache til andre operativsystemer. I tillegg er det også gjort endringer i modul API'en for lettere å kunne innføre nye funksjoner i fremtiden, uten å miste kompatibiliteten med gamle moduler.

Som nevnt i avsnittet om arkitektur (3.2.2), håndterer Apache spørringer i enkelt-trådet modus. På mange operativsystem kunne man oppnådd langt bedre ytelse ved å kjøre prosesser i fler-trådet-modus. Dette er imidlertid ikke mulig i nåværende versjon. Å bruke flere prosesser, slik som i dag, har imidlertid også noen fordeler; blant annet blir serveren mer stabil siden det alltid vil være andre prosesser som kan ta over i tilfelle en feiler. Hvis man hadde brukt en modell med flere tråder fordelt utover flere prosesser kunne man kanskje dra fordelen av både å ha en stabil og samtidig effektiv fler-trådet server. Dette har vært vanskelig å få til i den gamle serveren, spesielt fordi måten man mapper spørringer til prosesser ikke enkelt lar seg skrive om til å samtidig støtte tråder. I Apache 2 har man tatt konsekvensen av dette. Serveren vil nå få en ny hybrid løsning som gjøre det mulig å kjøre flere prosesser i fler-tråd-modus. En ny type pluggbare moduler kalt "Multiple-Processing Module," eller MPM vil bli tatt i bruk. Én Apache server vil få én MPM-modul. Denne modulen tar ansvar for ekspedering av spørringer og oppstart og stopping av prosesser og tråder. MPM-modulen beskriver med andre ord fullstendig hvordan serveren skal håndtere prosesser og tråder. På denne måte vil man kunne lage moduler med spesielle egenskaper tilpasset hvert enkelt operativsystem. Windows NT vil for eksempel få en MPM-modul som bruker en én-til-mange prosess/tråd-modell. På denne måten blir det mulig å yteses-optimalisere mange operativsystemer på en strukturert måte.

En annen teknikk som vil bli introdusert med Apache 2 og som også muliggjør operativsystem-avhengige optimaliseringer, er innføringen av et nytt abstraksjonslag i kildekoden. Man har kalt dette "Apache Portable Run-Time", eller APR. Idéen er å gi modulutviklere på Apache tilgang til et sett abstrakte funksjoner. Under kompilering kobles disse abstrakte funksjonene til

operativsystem-unike funksjoner som ofte er mer optimalisert. For eksempel brukes POSIX-funksjoner i nåværende versjon av Apache for Windows. Dette er ikke like effektiv som om man hadde benyttet seg av Windows egne proprietære API'er. Med APR blir det lettere å lage versjoner av Apache for operativsystemer som ikke støtter POSIX.

3.2.6 Dynamisk innhold

Den viktigste konsekvensen Shambhala API'en har hatt for Apache var at den gjorde det mulig å plugge inn moduler som kunne levere dynamisk innhold. Det vil si at en algoritme i modulen genererer en skreddersydd respons for hver spørring modulen tar ansvar for å betjene. Innholdet som returneres til klienten vil som regel være en HTML-fil, men det er også mulig å danne andre filtyper som for eksempel bilder eller video. Når en dynamisk web-side genereres er det ofte vanlig at modulen gjør spørringer til en database og fletter disse opplysningene inn i dokumentet som returneres til klienten. Tabell 3.4 viser de mest brukte modulene som kan levere dynamiske sider fra Apache.

Tabell 3.4 Moduler i Apache som genererer dynamiske websider

Modul navn	Beskrivelse
mod_cgi	Det var med CGI-skriptene (Common Gateway Interface) at dynamisk web innhold først slo gjennom. Med CGI kunne man starte programmer og sende resultatene fra disse tilbake til nettleseren.
mod_perl	Perl skript (Practical Extraction and Report Language) kjørtes tidligere gjennom CGI-grensesnittet, men på grunn av det noe dårlige CGI-designet har man i dag heller innført en egen modul for Perl-skript.
mod_php	PHP (gammel fork.: "Personal Home Pages", ny fork.: "PHP: Hypertext Preprocessor") er et skriptspråk med kommandoer som er innskutt i vanlige HTML-dokumenter. Før HTML-dokumentet sendes til klienten kjøres det gjennom en tolker som utfører hver PHP-kommando og erstatter den med sitt eget resultat. PHP har gode muligheter for databasetilkobling.
mod_jserv	Støtter Java servlets og JSP (Java Server Pages). JSP er tolket HTML-kode (á la PHP)
mod_java	En modul som gjør det mulig å utvikle Apache moduler skrevet og kjørt i Java. Modulen er ikke spesielt beregnet for å produsere dynamisk innhold, men er tatt med i denne tabellen for ordens skyld.
	ASP (Active Server Pages) er tolket HTML-kode (á la PHP). ASP har blitt veldig populært på Windows plattformen.

3.3 Java

Java er et C/C++ inspirert programmeringsspråk, utviklet spesielt med tanke på Internett. I likhet med C++ er Java også objektorientert, men siden Java er designet uten krav til kompatibilitet med elder språk, er det også gjennomført på en mer elegant måte. Dette har gjort Java til et brukervennlig og effektivt verktøy for utviklere.

Java har fått bred støtte i markedet og er i dag blitt en de facto standard. Selv om Java fremdeles er kontrollert og eid av Sun Microsystems, går utviklingen i retning av en mer åpen standard for Java. Spesielt gjelder dette mange av komponentarkitekturene som er utviklet for Java.

3.3.1 Java Virtual Machine, JVM

Kildekode i Java kompileres til såkalt bytekode. Bytekoden kjøres på en virtuell maskin som fungerer som et abstraksjonslag mellom Java programmene og operativsystemet. Et Java program kan derfor kjøres på alle datamaskiner som har en Java Virtual Machine (JVM). Dette gjør at Java programmer er "99 prosent" portable. Den siste prosenten skyldes ulikheter i noen av de virtuelle maskinene som finnes på markedet.

Gjennom JVM mappes Java funksjonalitet til operativsystemavhengig funksjonalitet. For eksempel vil en Java socket bli mappet til en socket i operativsystemet. Tilsvarende gjøres med trådene i Java. Dette er imidlertid litt mer komplisert. En tråd i Java kan mappes på to måter; enten som en "native thread" eller som en "green thread". Hvis en JVM er konfigurert til å bruke "native threads" vil en tråd i et Java program bli mappet til en ny tråd i JVM-prosess på operativsystemet. En JVM kan også settes opp til å selv utføre trådbehandlingen, uavhengig av om operativsystemet har trådstøtte eller ikke. Slike tråder som er støttet internt i JVM er ofte omtalt som "green threads". På denne måten er det mulig å kjøre Java tråder i JVM som går på operativsystemer uten trådstøtte.

3.3.2 Java Native Interface, JNI

Java Native Interface (JNI) ble laget for å gjøre det mulig å integrere C-funksjoner i Java programmer. Flere JVM leverandører hadde tidligere innført egne grensesnitt som kunne gjøre kall til C-kode, men disse manglet imidlertid en felles standard. For å løse dette problemet utviklet Sun JNI-standarden.

Det er hovedsakelig to grunner til at noen vil integrere maskinavhengige programmer i Java kode.

- Et program som er skrevet i C kan lettere optimaliseres for å oppnå bedre effektivitet enn det som er mulig i et Java program. Med JNI er det mulig å skrive Java programmer med tidskritiske funksjoner som er programmert i C.
- Mange organisasjoner sitter inne med store IT systemer som man har investert mye ressurser i for å utvikle. Disse kan ikke uten videre byttes ut med Java basert software. Hvis man ønsker å utvikle nye funksjoner for slike systemer er Java til C grensesnitt som JNI eneste utvei.

Før man begir seg ut med JNI bør man være klar over en del problematikk som kommer med teritoriet. For det første mister man mye av portabiliteten i Java. Programmet blir knyttet til en bestemt plattform og kan ikke uten videre flyttes til en annen. For det andre er det ikke alltid at JNI er den rette løsningen for å oppnå større effektivitet. For eksempel er et kall til C fra Java kjent for kunne være opp mot fem ganger så kostbar som en Java til Java interaksjon. Dette skyldes hovedsakelig at Java typer er representert på en annen måte enn C typer. Java bruker for eksempel 16 bits Unicode i tekststrenger mens C bruker 8 bits ASCII. Et kall til en C funksjon med en streng som parameter må derfor gjøre en ganske omfattende konvertering.

3.3.3 Java Servlets

En servlet er et program som er implantert i en webserver og som kan levere dynamisk innhold til nettlesere. Servleten har en egen ressurs identifikator på webserveren slik at nettlesere kan adressere servleten ved hjelp av en URL. Med "dynamisk" menes at innholdet er generert av en algoritme i servleten som kjøres hver gang en nettleser gjør en spørring. En servlet brukes vanligvis for å generere en skreddersydd HTML-side, gjerne med informasjon som er flettet inn fra en database. Servlets kan også brukes til å lagre informasjon til databaser. Dette kan for eksempel være opplysninger som en bruker har fylt inn i et HTML-skjema og postet til webserveren som parametre sammen med en spørring (se tabell 3.1 GET og POST).

Sun har laget et rammeverk for hvordan en servlet og miljøet rundt servleten skal oppføre seg. Dette omtales gjerne som Java Servlet Development Kit (JSDK). De viktigste begrepene i dette rammeverket er oppsummert i tabell 3.5.

Tabell 3.5 Terminologi brukt i servlet rammeverk

Begrep	**Beskrivelse**
Servlet	Svarer på spørringer og genererer innhold som til slutt bringes tilbake til nettleseren.
Servlet Container	Sørger for grunnlegende funksjonalitet for at servlets skal kunne motta spørringer og sende respons. Servlet containere er ofte synonymt med servlet motor.
Servlet Context	Definerer en servlets syn på en webapplikasjon. Servlet kan lagre parametre i konteksten som senere kan brukes av andre servlets i webapplikasjonen. Det eksisterer en Servlet context per webapplikasjon.
Request	Et request objekt brukes som parameter til en metode i servleten som har ansvar for å betjene en spørring. Request objektet inneholder for eksempel informasjon om nettleserens IP adresse og post parametre som er sendt i spørringen.
Response	Et respons objekt brukes som parameter til en metode i servleten som har ansvar for å betjene en spørring. Response objektet brukes av servleten for å sende data tilbake til nettleseren.
Session	Definerer en servlets syn på en sesjon med en nettleser. En sesjon har en livstid som strekker seg over flere spørringer. På denne måten kan servlets lagre tilstandsinformasjon om en klient.
Java Servlet Pages	Java Servlet Pages er en spesiell form for servlet. JSP er et skriptspråk med Java kommandoer som er innskutt i vanlige HTML-dokumenter. Før HTML-dokumentet sendes til klienten kjøres det gjennom en tolker som utfører hver Java-kommando og erstatter den med sitt eget resultat.

En webapplikasjon er består av et sett med Servlet objekter og et Context som beskriver applikasjonen. En webapplikasjon kan enten kjøres på én JVM eller den kan være merket som distribuerbar. Med en distribuerbar webapplikasjon kan man spre servlets over flere JVM'er som enten kjører på samme vertsmaskin eller er distribuert over flere forskjellige vertsmaskiner.

Webapplikasjoner som er distribuerte har et Context objekt per JVM. Spørringer som er assosiert med en brukersesjon i en distribuert webapplikasjon må kun håndteres av én JVM. Dette er gjort for å løse noe av samtidighetsproblematikken rundt Session objektet og for å sikre at data forblir konsistent.

For å sikre at en webapplikasjon er skalerbar må alle parameterobjekter som lagres i et Session objekt, implementere Serializable grensesnittet. På denne måten kan servletcontaineren gjøre last balansering ved å flytte Session objektet fra en JVM til en annen.

Det viktigst å ta med seg fra dette avsnittet er at tilstandsinformasjon i en webapplikasjon lagres i ett Context objekt og flere Session objekter. Synkronisering av tilstandsinformasjon mellom alle JVM er viktig. Context objektet trenger kun en løs synkroniseringsmekanisme, mens Session objektet trenger streng synkronisering og det er derfor ikke tillate å distribuere Session objekter.

3.4 Oppsummering

Dette kapittelet har gitt en presentasjon av de viktigste teknologiene som danner grunnlaget for denne hovedoppgaven. Kapittelet har forsøkt å fokusere på de egenskapene ved teknologiene som er mest relevante i forhold til oppgaven.
I avsnitt 3.1 ble det sagt at World Wide Web er bygget på en klient-server arkitektur. Kommunikasjon mellom klient og server skjer ved hjelp av en protokoll, HTTP, som er lagt på toppen av TCP/IP.

I avsnitt 3.2 gav en omfattende innføring i webtjeneren Apache. Her ble det sagt hvordan Apache håndterer prosesser og tråder og hvordan serveren kan konfigureres ved hjelp av direktiver. Det ble også vist hvordan funksjonaliteten i Apache kunne utvides ved hjelp av pluggbare moduler.
Det siste avsnittet, 3.3 handlet om Java og Java Virtual Machine (JVM). Her ble det også gitt en innføring i Java Native Interface (JNI), som er et grensesnitt mellom Java programmer og C programmer. Sist i avsnittet ble det vist hvordan rammeverket til en servlet fungerer.

4. Java servlet arkitekturer

I de fire første avsnittene av dette kapittelet presenteres alternative måter å integrere Java funksjonalitet i en webserver. Avsnitt 4.1 beskriver det designet som trolig er mest populært i dag, og som brukes i ApacheJServ. I denne løsningen er JVM plassert i en egen prosess og kommuniserer over TCP med et modul i webtjeneren. Avsnitt 4.2 gir et forslag til et alternativt design. Idéen er enkel; erstatt TCP-socket grensesnittet med en kommunikasjons API som er basert på Java Native Interface. Avsnitt 4.3 viser en løsning der webserveren er skrevet i Java, og i avsnitt 4.4 et design som egentlig ikke innbefatter Java servlets, men som likevel tillater at man bruker Java kode for å levere et dynamisk innhold fra en webserver. Etter gjennomgangen av alle arkitekturene følger avsnitt 4.5 opp med en kvalitativ sammenligning og konklusjon. Kriteriene som brukes i den følgende vurderingen er: skalerbarhet, effektivitet og transaksjonskostnad, sikkerhet, stabilitet og feiltoleranse, kompleksitet og brukervennlighet og til slutt portabilitet.

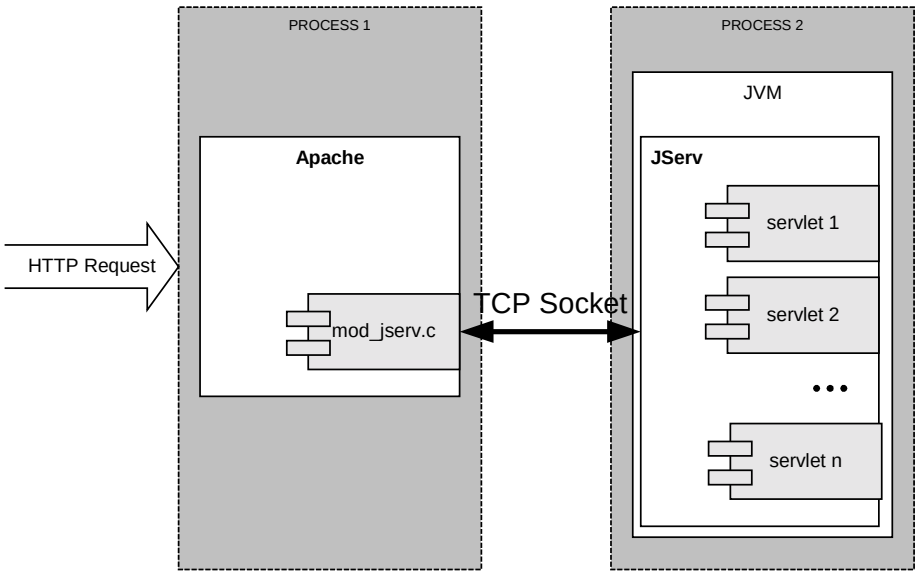
Avsnitt 4.5 gir en helhetlig vurdering og sammenligning av arkitekturene, og i avsnitt 4.6 argumenteres det for hvorfor det ikke er gjort en kvantitativ ytelsessammenlikning. Til slutt kommer et sammendrag av kapittelet i avsnitt 4.7

4.1 Servlet i selvstendig prosess (JServ)

En webserver som ikke er skrevet i Java, men som skal integrere Java servlets, trenger en eller annen mekanisme for å kommunisere med den virtuelle Java verdenen. Det trolig mest brukte designet i denne sammenheng er å kjøre en JVM i en prosess som er adskilt fra webserver og som kommuniserer over en IPC-mekanisme, vanligvis en TCP socket forbindelse. Apache JServ har brukt denne metoden med gode resultater.

4.1.1 Designskisse

Figur 4.1 viser hvordan JServ kan integreres med en webtjener. I figuren er det brukt en Apache webserver, men eksempelet er generelt og kunne like gjerne vært brukt på andre webservere med plugginn arkitektur. Eksempelet viser to prosesser; Apache prosessen og JVM prosessen. Kommunikasjon mellom webtjeneren og servlet containeren skjer over en TCP socket forbindelse. I ApacheJServ er det definert en egen protokoll for hvordan marshalling skal gjøres og data serialiseres før det sendes over forbindelsen. Denne protokollen kalles ApacheJServ Protocol (AJP). Begge sider må implementere AJP. I Apache serven er dette gjort ved å sette inn en plugginn modul, mod_jserv, som er konform med Apaches Shambhala API (se avsnitt 3.2.4). Når en spørring mottas av Apache, vil mod_jserv fange opp denne og sende den videre til JServ-containeren med den rette servleten. Her produseres et innhold som etterpå bringes tilbake til webtjerneren. Apache serveren genererer den endelige responsen som sendes tilbake til nettleseren.



Figur 4.1 Apache JServ arkitektur

Hovedtyngden av implementasjonen i Apache JServ ligger i Java implementasjon av servlet containeren. mod_jserv er kun en lettvekts modul som har i oppgave å serialisere spørringer fra klienter og deserialisere responser fra JServ. TCP-socket forbindelsen som brukes i JServ settes opp når serveren startes og gjenbrukes i alle senere spørringer.

4.1.2 Fordeler og ulemper

Under følger en punktvis vurdering av JServ-arkitekturens egenskaper. Det er forsøkt å fremheve fordeler og ulemper som er relevante i forhold til de andre arkitekturene i de neste hovedavsnittene. Hvert punkt blir introdusert med en kort beskrivelse av hvilken betydning som er ilagt egenskapen som er vurdert.

• **Skalerbarhet** - *Med skalerbarhet forståes løsningens mulighet til å kunne opp-skaleres for å håndtere trafikk fra flere brukere.*
JServ arkitekturen er meget skalerbar. Webtjeneren og servletcontaineren kjører i separate prosesser, og kommunikasjonen mellom dem er basert på TCP sockets. Det er derfor mulig å flytte Java prosessen til andre maskiner for å avlaste webtjeneren. Apache JServ tillater også å kjøre mer enn en JVM. Man kan på denne måten opprette en klynge av maskiner som man sprer servlet containere utover på. Slik er det også mulig å innføre last balansering. Flaskehalsen vil til slutt bli selve Apache serveren, men denne kan også skaleres opp ved å bruke klynger av Apache servere i kombinasjon med for eksempel en IP-sprayer som fordeler spørringer utover flere Apache servere.

• **Effektivitet og transaksjonskostnader** - *Med et effektivt design menes for eksempel at koden er skrevet i et effektivt programmeringsspråk, slik som C, eller at det er programmert på en slik måte at software tar minst mulig ressurser fra maskinvaren den kjøres på. Med transaksjonskostnader forståes prosesseringstiden og de totale belastningene en server påføres som direkte årsak av transaksjonens prosessering og kommunisering.*

JServ må serialisere/deserialisere spørringer og responser for at mod_jserv og servletcontaineren skal kunne kommunisere sammen. Dette er med på å øke transaksjonskostnaden sammenlignet med andre design som ikke trenger å sende data over en link. JServ løsningen bruker TCP-sockets. En slik forbindelse kan ha høyere kommunikasjonskostnader forbundet med seg enn andre og mer effektive former for IPC (InterProcess Communication). For eksempel er TCP-sockets kjent for å være mer kostbare enn UNIX-sockets. (Merk.: Noen operativsystem tar imidlertid i bruk UNIX sockets bak i kullisene hvis en TCP socket åpnes på samme maskin).

• **Sikkerhet** - *Med sikkerhet menes hele systemets evne til å motstå forsøk fra uvedkommende på å trenge seg inn i systemet, enten for å observere, påvirke eller ødelegge data eller funksjonalitet. Sikkerhet i denne sammenhengen betyr ikke autentisering av brukere eller kryptering av pakker som sendes mellom nettleser og webtjener.*
Sikkerheten i Apache serveren er regnet for å være bra. Siden alle spørringer som sendes inn i løsningen må passere gjennom Apache, vil denne fungere som en slags brannmur. Selv om AJP-protokollen ikke er kryptert er som regel ikke dette noe problem. Trafikken mellom servletcontaineren og webserveren går i de fleste tilfeller kun innenfor et lite subnett som tjenestetilbyderen har full kontroll over, og uvedkommende derfor ikke vil få tilgang til.
Protokollen har også en enkel autentiseringsmekanisme (hemmelig nøkkel fil), men denne fungerer bare tilfredsstillende hvis servletcontaineren kjører på samme maskin som webserveren. Det betyr at dersom en servletcontainer kjøres på en annen maskin vil det være en mulighet for at personer med uredelige hensikter kan sende kommandoer til servletcontaineren og påvirke denne i negativ retning. Hvis servlet motoren kjøres på en annen maskin en den som Apache kjøres fra vil det være en fordel å bruke en brannmur.

• **Stabilitet og feiltoleranse** - *Med stabilitet menes hvor stor andel av driftstiden løsningen faktisk fungerer slik den skal. Med feiltoleranse menes hvor robust løsningen er når det gjelder å komme seg ut av situasjoner som kan utvikle seg til driftsbrudd.*

Et problem med de tre andre arkitekturene er at webserver og servlets kjører i samme prosess. Hvis en servlet krasjer kan dette føre til at hele webserveren havarerer. Feiltoleransen i JServ er litt bedre siden servletcontaineren kjører i en prosess som er adskilt fra Apache. En servlet kan på denne måten ikke alene ta ned hele webserveren. Under JServ kjøres Apache i normalt multiprosess modus. Dette er ikke tilfelle for JNI løsningen i avsnitt 4.2 eller Java webserver løsningen i avsnitt 4.3. Disse kjører som enkel prosess / multitråd. JServ vil derfor kunne være mer feiltollerant enn disse løsningene.

• **Kompleksitet og brukervennlighet** - *Med kompleksitet menes hvor sammensatt og vanskelig løsningen er å installere, konfigurere og vedlikeholde.*

JServ er i likhet med flere av de andre arkitekturene i dette kapittelet, konseptuelt sett ganske kompliserte. En administrator som bruker JServ må ha klart for seg hvordan nettleser, Apache, mod_jserv, JServ og JVM fungerer og kommuniserer sammen. I tillegg må han også ha kunnskap om hvordan servletcontaineren skal konfigureres.

• **Portabilitet** - *Med portabilitet menes hvor enkelt eller vanskelig det er å bringe arkitekturen over til andre operativsystemer uten å måtte gjøre store endringer i implementasjonen og API'ene.*

JServ er relativt enkelt å porte til andre operativsystem. Servletcontaineren kan kjøres på alle operativsystemer som har et Java miljø. Plugginn modulen for Apache, mod_jserv, er skrevet i vanlig ANSI C og lar seg derfor porte til alle operativsystemer som Apache støtter. Det er også mulig å lage plugginn moduler til andre webservere enn Apache.

4.1.3 Bruk av JNI for å effektivisere kommunikasjon

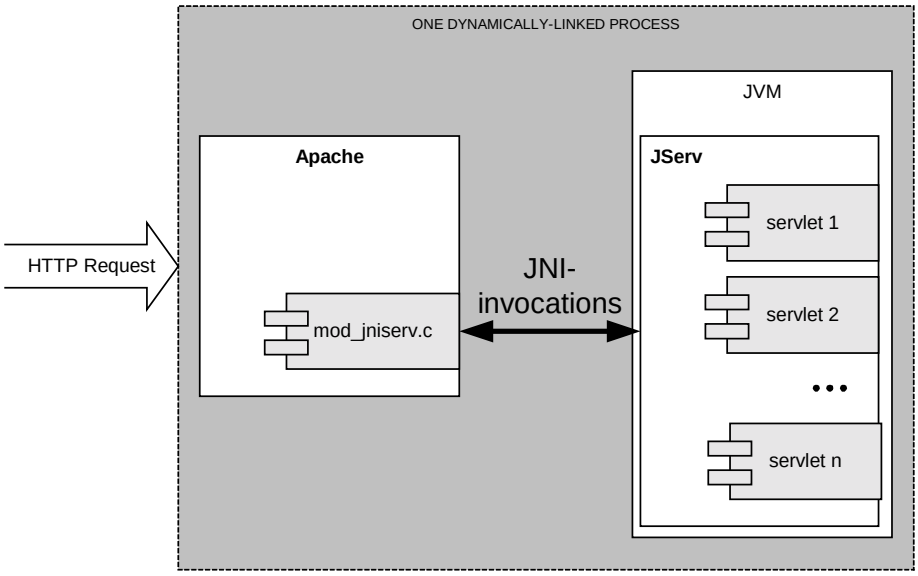
I forrige avsnitt ble det nevnt at transaksjonskostnadene i JServ arkitekturen kunne være høye fordi serialisering / deserialisering tar mye prosessortid. Siden disse operasjonene kan gjøres mer effektive i C enn i Java, kunne en måte å optimalisere kommunikasjonsflyten mellom JServ og mod_jserv vært å ta i bruk støttebiblioteker programert i C. Disse kunne være knyttet opp mot Java metoder i JServ ved hjelp av JNI. All kommunikasjon kunne på denne måten vært håndtert av prosessoroptimalisert kode. Egenskapene til et slikt design ville vært identisk med dagens JServ, med unntak at man måtte ha kompilert det dynamiske linkede biblioteket for de plattformene servlet containeren skulle kjøre på. For å sikre at løsningen ble like portabel som JServ ville det vært viktig å ikke bruke plattform spesifikk kode i støttebiblioteket, selv om dette kunne gjort løsningen mer effektiv.

4.2 Servlet i webserver prosess (JNIServ)

I forrige avsnitt ble det presentert en metode for å integrere Java servlets i en webtjener og en løsning, kalt JServ, som implementerer denne metoden. JServ fungerer ved at en plugginn modul (mod_jserv) i Apache serveren kommuniserer med en servletcontainer over en TCP-socket forbindelse. Servletcontaineren kjøres på en JVM-prosess som er adskilt fra Apache prosessen. I avsnitt 4.1.2 ble det blant annet pekt på at denne løsningen ikke var veldig effektiv. Prosessen med å serialisere/deserialisere og sende data over TCP-socket var tids- og ressursmessig kostbar. Etter at JNI grensesnittet ble tatt i bruk, har det blitt mulig å integrere JVM i egenutviklede programmer. En idé kunne derfor være å integrere de to prosessene i JServ og kjøre den virtuelle maskinen fra samme prosess som Apache. Siden denne løsningen p.t. ikke er tatt i bruk, har jeg selv funnet et passende navn; "JNIServ".

4.2.1 Designskisse

Figur 4.2 viser et design med JVM integrert i samme prosess som webtjeneren. I motsetning til JServ er kommunikasjonen med Java gjort gjennom JNI-kall. Ved oppstart av Apache vil mod_jniserv sørge for å laste JVM fra et bibliotek som er linket inn i modulen. Resten av oppsettet fungere tilsvarende som for JServ, men det er likevel store forskjeller mellom de to designene. Disse forskjellene vil bli fremhevet i neste avsnitt om fordeler og ulemper.



Figur 4.2 Webserver og JVM i samme prosess

4.2.2 Fordeler og ulemper

Under følger en punktvis vurdering av denne arkitekturens egenskaper. Det er forsøkt å fremheve fordeler og ulemper som er relevante i forhold til de andre arkitekturene. Hvert punkt blir introdusert med en kort beskrivelse av hvilken betydning som er ilagt egenskapen som er vurdert.

• **Skalerbarhet** - *Med skalerbarhet forståes løsningens mulighet til å kunne opp-skaleres for å håndtere trafikk fra flere brukere.*
JNI-løsningen skalerer svært dårlig. I avsnitt 3.2 ble det nevnt at Apache kjører flere parallelle prosesser for å kunne håndtere mange samtidige spørringer. Disse prosessene er forgrenet fra en root prosess som overvåker antall ledige tjenerprosesser. Root prosessen starter eller avslutter tjenerprosesser for hele tiden å kunne opprettholde et optimalt system med hensyn til belastning og responstid. I en slik prosessmodell vil det være vanskelig å introdusere en servletmotor basert på JVM og JNI. Hvis man likevel velger å gå for en slik løsning vil to alternativer være aktuelle;

- Kjøre en JVM i hver av Apaches tjenerprosesser. Dette innebærer en rekke problemer. Hver inkarnasjon av JVM trekker minne i størrelsesorden fra en halv til titalls megabyte. Minneforbruket vil øke proporsjonalt med antall tjenerprosesser og minne forbruket i en server som behandler mange spørringer vil bli ekstremt.

Et annet problem som dukker opp i denne løsningen er hvordan tilstandsproblematikken i forbindelse med servlets skal håndteres. Siste versjon av Java Servlet Specifiaction v2.2 [14] har innført noen som kalles en distribuert webapplikasjon. Hvis denne løsningen skal kunne fungere må alle webapplikasjoner gjøres distribuerbare slik som beskrevet i denne spesifikasjonen. Dette vil komplisere arbeidet for utviklere siden det da må taes spesielle hensyn ved utvikling av servlets (se avsnitt 3.3.3).

En bruker av en webapplikasjon må på en eller annen måte gjenkjennes fra spørring til spørring og sendes til den JVM hvor brukerens Session objekt befinner seg. Når en spørring ankommer Apache vil denne bli sendt til første og beste tjenerprosess. Sannsynligheten for at spørringen da har blitt sendt til feil tjenerprosess og JVM er stor. Servlet motoren kan da bare gjøre en ting; sende spørringen videre til den tjenerprosess og JVM som brukerens Session er lagre på.

På denne måten går vinningen (med JNI i samme prosess som Apache) opp i spinningen.

Apache 2 har en mer avansert modell for hvordan spørringer fordeles til prosesser (se avsnitt 3.2.5 om MPM-moduler). Hvis det hadde vært mulig å lage en MPM modul som klarer å sende spørringen til riktig tjenerprosess/JVM på første forsøk ville dette problemet vært løst. Jeg har imidlertid ikke fått verifisert om dette er mulig.

- Over ble det foreslått å kjøre en JVM i hver av Apaches tjenerprosesser. Et alternativ til dette er å kjøre Apache i én prosess og kun bruke én JVM. Mye av essensen og effektiviteten til Apache faller imidlertid da bort siden UNIX versjonene av Apache kjøres i enkeltrådet modus og kun en bruker vil da kunne betjenes av gangen. Det er kun Windows versjoner som i dag kjører i fler-tråd-modus og i én prosess. For å få dette til på UNIX må man smøre seg med tålmodighet og vente til Apache 2.0 er ute.

Konklusjonen her er at JNI-løsningen skalerer svært dårlig i dagens versjoner av Apache. Hovedproblemet er at man enten må kjøre en JVM for hver Apache tjenerprosess, eller at man må nedskalere Apache til kun å kjøre i en prosess. Det er også skissert en måte mange av disse problemene kanskje kan ungås i neste generasjon av Apache.

En annen faktor som er med på å redusere skalerbarheten er at det ikke er mulig å legge servletmotoren på en annen maskin hvis man bruker en modul som utelukkende kommuniserer ved hjelp av JNI. Et løsningforslag kunne være å lage en hybrid modul som støtter både JServ designet og JNI designet. Brukeren kunne på denne måten valgt hva som passet best, for eksempel ved å sette direktiver i konfigurasjonsfilene.

• **Effektivitet og transaksjonskostnader** - *Med et effektivt design menes for eksempel at koden er skrevet i et effektivt programmeringsspråk, slik som C, eller*

at det er programmert på en slik måte at software tar minst mulig ressurser fra maskinvaren den kjøres på. Med transaksjonskostnader forståes prosesseringstiden og de totale belastningene en server påføres som direkte årsak av transaksjonens prosessering og kommunisering.
Det er først og fremst på grunn av et håp om bedre effektivitet at JNI-løsningen er tatt opp til vurdering. I teksten over ble det argumentert for at en god skalerbarhet og effektivitet vil være vanskelig å få til med JNI-designet hvis dette skulle implementeres i dagens versjon av Apache. Samtidig ble det pekt på noen måter som dette kanskje kan løses på i Apache 2.0. Det kunne derfor vært interessant å se på hvordan man kan gjøre selve implementasjonen så effektiv som mulig under disse forutsetningene.

Et av valgene som må taes er hvordan man vil implementere servlet motoren. To alternativer kan være aktuelle:
- Implementere hele servletcontaineren i C-kode og lage en Java innpakning av disse. På denne måten kunne man fått en høyt optimalisert C-basert servletengine. En faktor som imidlertid vil trekke ned ytelsen er hvor mange JNI-interaksjoner som er inne i bildet. Selv om JNI interaksjonene nok er en god del raskere enn IPC mekanismene i JServ, er det viktig å være klar over at det også er ganske høye kostnader forbundet med JNI. Et kall fra Java til en C-funksjon er påstått å typisk være fem ganger mer kostbart enn et Java til Java kall. Det er med andre ord viktig å redusere antall JNI interaksjoner til et minimum. En servlet gjør mange interaksjoner med Request og Response objekter. Hvis disse kallene kan fort bli ganske kostbare hvis de må gjøres over et JNI-grensesnitt.
- Et alternativ som derfor kanskje er mer effektivt er å legge hele implementasjonen av servletmotoren til Java. Man kunne da fått redusert antall JNI-interaksjoner ved kun å overbringe spørringen den ene veien og responsen den andre. Ulempen ved å kjøre servletmotoren i bytekode kunne på denne måten blitt hentet inn igjen.

• **Sikkerhet** - *Med sikkerhet menes hele systemets evne til å motstå forsøk fra uvedkommende på å trenge seg inn i systemet, enten for å observere, påvirke eller*

ødelegge data eller funksjonalitet. Sikkerhet i denne sammenhengen betyr ikke autentisering av brukere eller kryptering av pakker som sendes mellom nettleser og webtjener.

JNI-løsningen bruker ingen IPC eller RPC mekanismer som lett lar seg utsette for inntrenging. Webtjeneren og JVM kjøres innenfor samme prosess, og denne prosessen er i henhold til Apaches prosessmodel kjørt i et normalt begrenset brukermodus. Det er derfor å anta at sikkerheten er bedre i JNI-løsninger enn i JServ.

• **Stabilitet og feiltoleranse** - *Med stabilitet menes hvor stor andel av driftstiden løsningen faktisk fungerer slik den skal. Med feiltoleranse menes hvor robust løsningen er når det gjelder å komme seg ut av situasjoner som kan utvikle seg til driftsbrudd.*
Ved å integrere en JVM maskin i samme prosess som Apache er nødvendigvis ikke en god idé hva stabilitet angår. En JVM er en meget komplisert software. Sammen med JNI blir ikke løsningen mindre kompleks. I et slikt system kan minnelekk fort bli et problem. Apache bruker en såkalt "resource alocation pool" men denne vil ikke JVM kunne ta del i.
JNI-løsningen har derfor sannsynligvis lettere for å bli ustabil enn JServ-løsningen. Hvis hele webtjeneren kjøres under en prosess vil et krasj i JVM kunne ta ned hele webtjenesten.

• **Kompleksitet og brukervennlighet** - *Med kompleksitet menes hvor sammensatt og vanskelig løsningen er å installere, konfigurere og vedlikeholde.*
JNI-løsningen innfører som beskrevet over masse ny problematikk som webadministratoren må vite om for å kunne konfigurere Apache slik at modulen skal kunne fungere.

• **Portabilitet** - *Med portabilitet menes hvor enkelt eller vanskelig det er å bringe arkitekturen over til andre operativsystemer uten å måtte gjøre store endringer i implementasjonen og API'ene.*

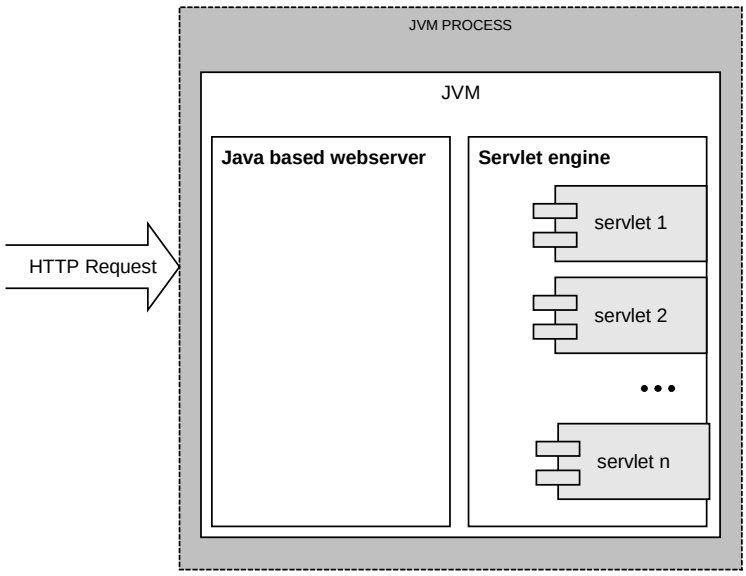
JNI-løsningen er vanskelig å porte til operativsystemer som ikke har innebygget trådstøtte. Hvis operativsystemet kun kan kjøre webtjenere i uavhengige prosesser vil det være nødvendig å kjøre en JVM per webtjener prosess. Dette kan gå hardt utover minne og er derfor ikke en gunstig løsning.

4.3 Servlet i JavaWebServer

I avsnitt 4.1 og 4.2 har det vært vis eksempler på arkitekturer hvor webtjeneren har vært skrevet i C-kode. I dette avsnittet vurderes en løsning som er totalt basert på Java.

4.3.1 Designskisse

Figur 4.3 viser en JVM som kjører en webserver med en integrert servletcontainer. Spørringer fra nettlesere mottas direkte av JVM og hele transaksjonen prosesseres i Java.



Figur 4.3 Java basert webserver med servlets

4.3.2 Fordeler og ulemper

Under følger en punktvis vurdering av denne arkitekturens egenskaper. Det er forsøkt å fremheve fordeler og ulemper som er relevante i forhold til de andre arkitekturene. Hvert punkt blir introdusert med en kort beskrivelse av hvilken betydning som er ilagt egenskapen som er vurdert.

• **Skalerbarhet** - *Med skalerbarhet forståes løsningens mulighet til å kunne opp-skaleres for å håndtere trafikk fra flere brukere.*
JavaWebServer skalerer bra. Selv om den Java baserte webserveren kjører mindre effektivt enn en Apache server, vil det likevel være mulig å skalere opp ved å ta i bruk flere parallelle servere. Spørringer fra klienter kan da for eksempel last balanseres ved hjelp av en ruter med IP-sprayer.

• **Effektivitet og transaksjonskostnader** - *Med et effektivt design menes for eksempel at koden er skrevet i et effektivt programmeringsspråk, slik som C, eller at det er programmert på en slik måte at software tar minst mulig ressurser fra maskinvaren den kjøres på. Med transaksjonskostnader forståes prosesseringstiden og de totale belastningene en server påføres som direkte årsak av transaksjonens prosessering og kommunisering.*
Det er klart at en webserver basert på Java vil stille i en annen klasse enn en C-kodet og optimalisert webserver som Apache. Responstiden fra denne serveren er stor og effektiviteten er ganske dårlig.

• **Sikkerhet** - *Med sikkerhet menes hele systemets evne til å motstå forsøk fra uvedkommende på å trenge seg inn i systemet, enten for å observere, påvirke eller ødelegge data eller funksjonalitet. Sikkerhet i denne sammenhengen betyr ikke autentisering av brukere eller kryptering av pakker som sendes mellom nettleser og webtjener.*

Det er vanskelig å si noe generelt om sikkerheten i denne løsningen. Den største faren med løsningen er nok at hvis noen klarer å bryte seg gjennom sikkerheten i en av servletene så står hele webserveren åpen for angrep.

• **Stabilitet og feiltoleranse** - *Med stabilitet menes hvor stor andel av driftstiden løsningen faktisk fungerer slik den skal. Med feiltoleranse menes hvor robust løsningen er når det gjelder å komme seg ut av situasjoner som kan utvikle seg til driftsbrudd.*
Sammenlignet med Apache er sannsynligvis en JVM mindre stabil.

• **Kompleksitet og brukervennlighet** - *Med kompleksitet menes hvor sammensatt og vanskelig løsningen er å installere, konfigurere og vedlikeholde.*
Dette designet er konseptuelt enklere enn de tre andre arkitekturene som er vurdert i dette kapittelet. Derfor dette også den løsningen som har størst potensial for å bli enklest å bruke.

• **Portabilitet** - *Med portabilitet menes hvor enkelt eller vanskelig det er å bringe arkitekturen over til andre operativsystemer uten å måtte gjøre store endringer i implementasjonen og API'ene.*
En Java basert webserver kan portes til alle operativsystemer som har en Java Virtual Machine. Dette er derfor den mest portable løsningen i denne vurderingen.

4.4 Java men ikke servlet

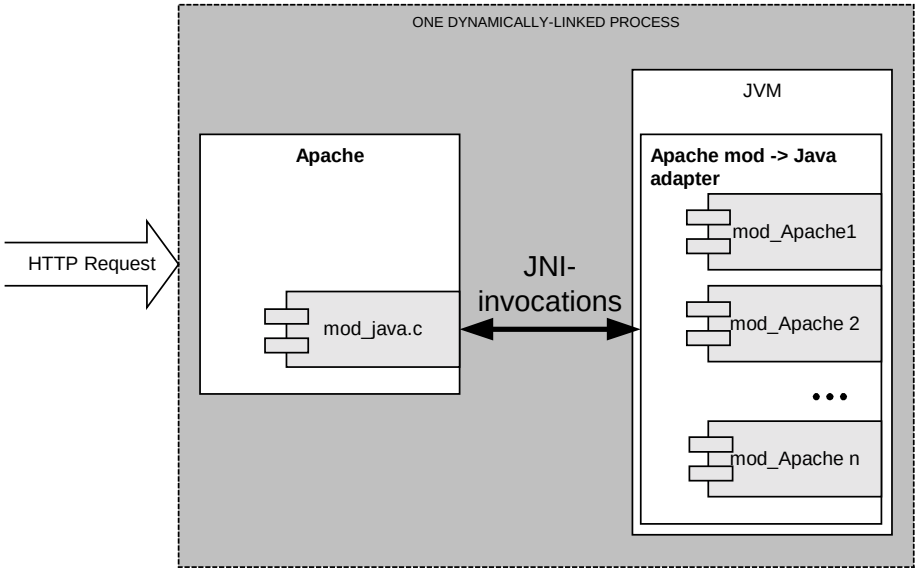
De første tre avsnittene i dette kapittelet har vist eksempler på arkitekturer som integrerer Java servlets i en webtjener. Dette avsnittet viser også en løsning som kan generere dynamisk innhold, men denne løsningen skiller seg fra de andre ved at den ikke følger rammeverket for servlets [14]. I steden har man innført en ny Java klasse som er konform med modul arkitekturen i Apache.
Det gjøres ingen vurdering av fordeler og ulemper av designet som er vist i dette avsnittet fordi løsningen skiller seg vesentlig fra de andre. Det kan likevel være

interessant å vite at det er andre måter å produsere dynamisk innhold med Java enn å bruke Java servlets.

4.4.1 Designskisse

Figur 4.4 viser hvordan det er mulig bruke et Java objekt som om det skulle vært en fullt utrustet Apache modul. Løsningen bruker en lettvekts modul, mod_java, som videresender all interaksjon med Shambhala API'en (se avsnitt 3.2.2) til et eller flere Java objekter som er konforme med denne API'en. På denne måten er det mulig å utvikle prototyp- eller full skala moduler på en hurtig og effektiv måte i Java. Løsningen kan også i noen tilfeller brukes i webapplikasjoner for å levere dynamisk innhold til klienter.

I denne løsningen er JVM integrert i samme prosess som Apache og interaksjon med Shambhala API'en mappes til Java kall gjennom Java Native Interface. Dette er en løsning som enda ikke er testet, men som en liten undergruppe i Apache Software Foundation [7] vurderer å lage.



Figur 4.4 Java baserte Apache moduler

4.5 Kvalitativ sammenligning

I dette avsnittet gjøres en oppsummering av fordeler og ulemper som er funnet ved de tre første arkitekturene i dette kapittelet.

4.5.1 Skalerbarhet

Med skalerbarhet forståes løsningens mulighet til å kunne opp- skaleres for å håndtere trafikk fra flere brukere.

- JServ er den løsningen som klart skalerer best fordi dette designet tillater å kjøre flere JVM på flere forskjellige vertsmaskiner samtidig.
- JNIServ har ikke noe potensial til å bli en brukbar løsning før Apache 2 er ute med fler-tråd støtte. Da vil situasjonen se veldig forskjellig ut.
- Det er også mulig å skalere opp JavaWebServer, men fordi ytelsen i Java implementasjoner er dårligere enn det den er i C, vil dette ikke være et reelt alternativ.

4.5.2 Effektivitet og transaksjonskostnader

Med et effektivt design menes for eksempel at koden er skrevet i et effektivt programmeringsspråk, slik som C, eller at det er programmert på en slik måte at software tar minst mulig ressurser fra maskinvaren den kjøres på. Med transaksjonskostnader forståes prosesseringstiden og de totale belastningene en server påføres som direkte årsak av transaksjonens prosessering og kommunisering.

- JServ får en ekstrakostnad i og med at den sender alle spørringer og responser over en socket.
- Ved å implementere JNIServ på riktig måte vil sannsynligvis dette bli den raskeste løsningen. Denne implementasjonsteknikken er dessverre ikke tilgjengelig før Apache 2 er sluppet.

• JavaWebServ lider fordi Java Virtual Machine ikke kan kjører Java program like effektivt som et program skrevet i C.

4.5.3 Sikkerhet

Med sikkerhet menes hele systemets evne til å motstå forsøk fra uvedkommende på å trenge seg inn i systemet, enten for å observere, påvirke eller ødelegge data eller funksjonalitet. Sikkerhet i denne sammenhengen betyr ikke autentisering av brukere eller kryptering av pakker som sendes mellom nettleser og webtjener.
Sikkerhetsproblematikken er egentlig ikke så forskjellig i de tre løsningene. JServ er kanskje litt mere utsatt siden all kommunikasjon gjøres over en TCP/IP socket som ikke er kryptert eller skikkelig autentisert.

4.5.4 Stabilitet og feiltoleranse

Med stabilitet menes hvor stor andel av driftstiden løsningen faktisk fungerer slik den skal. Med feiltoleranse menes hvor robust løsningen er når det gjelder å komme seg ut av situasjoner som kan utvikle seg til driftsbrudd.
I JNIServ og JavaWebServer kjøres webtjeneren og servlets i samme prosess. Dette gjøre at JServ, som har JVM i separat prosess, nok blir litt mer stabil.

4.5.5 Kompleksitet og brukervennlighet

Med kompleksitet menes hvor sammensatt og vanskelig løsningen er å installere, konfigurere og vedlikeholde.
Løsningene som inkorporerer en Apache server krever at webadministratoren sitter inne med mer kunnskap enn hva som er nødvendig med en server som utelukkende er basert på Java. Av denne grunn er det derfor JavaWebServer som er minst kompleks og mest brukervennlig.
De to andre løsningene, JServ og JNIServ, er stort sett identiske når det gjelder brukervennlighet.

4.5.6 Portabilitet

Med portabilitet menes hvor enkelt eller vanskelig det er å bringe arkitekturen over til andre operativsystemer uten å måtte gjøre store endringer i implementasjonen og API'ene.

- JavaWebServer er mest portabel
- JServ kan portes til alle Apache servere og servletcontaineren kan kjøre på de fleste maskiner.
- For JNIServ kan i realiteten kun kjøres på maskiner som har støtte for tråder.

4.5.7 Helhetsvurdering og konklusjon

Som en konklusjon på den kvalitative vurderingen av de tre arkitekturene som er vurdert er det mulig å si følgende:

- JServ er den best løsningen i dag.
- JNIServ er ikke et reelt alternativ før Apache 2 er sluppet. Da kan JNIServ sannsynligvis bli den løsningen som flest bør velge.
- ApacheWebServer er kun brukbar som webtjener for et lite antall brukere. Denne løsningen er den minst kompliserte og mest brukervennlige av de tre løsningene.

4.6 Kvantitativ sammenligning av ytelse

I oppgaveteksten (se vedlegg 2) for denne hovedoppgaven var det i 3. og 4. punkt satt opp et mål om å lage en implementasjon av den løsningen som i avsnitt 4.2 går under navnet JNIServ. Etter en litteratur- og teknologi studie og en kvalitativ vurdering (se foregående avsnitt i dette kapittel), ble det konkludert med at det ikke var hensiktsmessig å gjøre en implementasjon av JNIServ før versjon 2.0 av Apache var lansert. Etter undersøkelsen av de forskjellige teknologiene ble det også observert at det å gjøre en implementasjon av JNIServ ville bli ganske omfattende.

4.7 Oppsummering

I dette kapittelet er det presentert fire forskjellige design som gjør det mulig å lever dynamisk generert innhold fra en webtjener. I alle arkitekturene er det Java-algoritmer som produserer innholdet. Tre av metodene er basert på et rammeverk for Java servlets, mens den siste metoden har prøvet å etterligne modul API'en i Apache. Det er videre gjort en vurdering av de tre servlet arkitekturene og deres gyldighet i forskjellige situasjoner. Tabell 4.1 gir en oppsummering av de viktigste egenskapene til arkitekturene.

Den viktigste observasjonen som ble gjort under studiet var at det såkalte JNIServ-designet vanskelig lar seg forene med dagens utgave av Apache v1.3. Løsningen vil imidlertid kunne fungere bra sammen med den fremtidige Apache v2.0.

I avsnitt 4.6 er det argumentert for hvorfor det ikke var hensiktsmessig å implementere JNIServ i dag.

Tabell 4.1 Klassifisering av arkitekturer

	JServ	**JNIServ**	**JavaWebServer**
Skalerbarhet	Best i dag	Dårlig i dag, best i morgen	Skalerer, men er for ineffektiv til å brukes i større sammenhenger
Effektivitet	Best i dag	Dårlig i dag, best i morgen	Dårlig
Sikkerhet	Kommunikasjon over TCP/IP er ikke kryptert eller autentisert	Ok	Ok
Stabilitet	Best	Dårlig i dag, bedre i morgen	
Brukervennlighet	Ok	Ok	Best
Portabilitet	Ok	Lar seg bare porte til operativsystemer som støtter tråder.	Best

5. Konklusjon

Dette dokumentet ser nærmere på plugginn arkitektur for webservere. Jeg har spesielt sett på muligheten for å gjøre endringer i designet av "Apache JServ". Apache JServ er en servlet motor, det vil si en løsning som brukes for å knytte Java servlets til en web-tjener. Servlets i Apache JServ kjøres som alle andre Java programmer, i en virtuell maskin (JVM). Denne går i en egen prosess, adskilt fra web-tjeneren. Kommunikasjon mellom web-tjeneren og JVM / JServ gjøres ved hjelp av en TCP-socket forbindelse hvor man benytter en spesielt utviklet protokoll, kalt AJP. I webtjeneren setter man inn en pluggbar modul som implementerer AJP grensesnittet ut mot JServ. På denne måten er det mulig å lage moduler for alle typer webservere som på en eller annen måte støtter pluggbare komponenter. I dag finnes det kunne et modul, mod_jserv.c, for web-tjeneren Apache.

I oppgaven gjøres det en vurdering om det kan være hensiktsmessig å endre det TCP-socket baserte grensesnittet til JServ med et grensesnitt som er basert på Java Native Interface (JNI), og på denne måten integrere JVM i samme prosess som Apache. Dette alternative designet har i denne oppgaven blitt gitt arbeidstittelen "JNIServ".

Det trekkes frem både fordeler og ulemper både ved JServ og ved JNIServ. Den viktigste konklusjonen i arbeidet er imidlertid at det ikke er hensiktsmessig å implementere JNIServ før Apache versjon 2 er lansert. Dette fordi at Apache i dag har en låst måte å behandle prosesser og tråder og denne modellen passer dårlig sammen med JNIServ designet.

Litteraturliste

Bøker:

[1] P. Wainwright, *Professional Apache*, Wrox Press 1999 (ISBN 1-861003-02-1)

[2] M. J. Kabir, *Apache Server Bible*, IDG Books Worldwide 1998 (ISBN 0-7645-3218-9)

[3] R. Gordon, *Essential JNI - Java™Native Interface*, Prentice Hall PTR 1998 (ISBN 0-13-679895-0)

[4] G. Colouris, J. Dollimore, T. Kindberg, *Distributed Systems - Concepts and Design, Second Edition*, Addison-Wesley 1994 (ISBN 0-201-62433-8)

[5] P. Rossbach, H. Schreiber, *Java Server & Servlets - Building Portable Web Applications*, Addison-Wesley 2000 (ISBN 0-201-67491-2)

[6] L. L. Peterson, B. S. Davie, *Computer Networks - A Systems Approach*, Morgan Kaufmann Publishers 1996 (ISBN 1-55860-368-9)

Apache prosjektet på Internett:

[7] Apache Software Foundation (http://www.apache.org/)

[8] Java Apache Project (http://java.apache.org/)

[9] Apache Development Site (http://dev.apache.org/)

[10] Apache Module Registry (http://modules.apache.org/)

Java, Suns sider på Internett:

[11] Java at Sun (http://java.sun.com/)

[12] The Java Tutorial (http://java.sun.com/docs/books/tutorial/index.html)

[13] Java Platform Ports (http://java.sun.com/cgi-bin/java-ports.cgi)

[14] Java Servlet API (http://java.sun.com/products/servlet/index.html)

IETF, Internet Engineering Task Force: (http://www.ietf.org/rfc.html)

[15] T. Berners-Lee, D. Connolly, *Hypertext Markup Language - 2.0*, IETF november 1995 (RFC#1866)

[16] T. Berners-Lee, R. Fielding, H. Frystyk, *Hypertext Transfer Protocol -- HTTP/1.0*, IETF mai 1996 (RFC#1945)

[17] R. Fielding, J. Gettys m.fl., *Hypertext Transfer Protocol -- HTTP/1.1*, IETF juni 1999 (RFC#2616)

[18] N. Freed, N. Borenstein m.fl., *Multipurpose Internet Mail Extensions (MIME)*, IETF november 1996 (RFC#1945, #1946, #1947, #1948, #1949)

Andre sider på Internett:

[19] D. M. Epp, *Interfacing Java with C in Linux*, CScene nr. 4, 25 juni 1998 (http://cscene.org/CS4/CS4-01.html)

[20] A. D. Marshall, *Programming in C - UNIX System Calls and Subroutines using C*, mars 1999 (http://www.cm.cf.ac.uk/Dave/C/CE.html)

[21] SuSE Linux Inc. (http://www.suse.com/)

[22] Blackdown Java Linux Project (http://www.blackdown.org/java-linux.html)

[23] CERN, *Where the Web was born* (http://public.web.cern.ch/Public/ACHIEVEMENTS/web.html)

[24] Netcraft (http://www.Netcraft.com/)

[25] R. Bloom, *Apache 2.0 - The Next Generation*, Apache Week nr. 173 24.september 1999 (http://www.apacheweek.com/features/apache20)

Effektiv integrasjon av Java Servlets i web-tjener

Forslag til Diplomoppgave
Øyvind Hanssen, Agder IKT senter

1. Innledning

Vi skal belyse design og implementasjon av tjener-applikasjoner som er implementert i språk som C men hvor funksjonalitet kan utvides med dynamisk pluggbare komponenter skrevet i Java. Vi skal undersøke fordeler og ulemper ved en arkitektur hvor Java virtuell maskin (og java komponentene) kjører innenfor samme prosess som tjeneren, i forhold til en arkitektur hvor tjeneren og JVM kjører i separate prosesser og kommuniserer ved hjelp av operativsystemets IPC mekanismer (TCP sockets).

Mer konkret skal vi se på Java Servlets som en utvidelse av web-tjenere. Java servlets er tjener-motstykket til Applets. Servlets er Java komponenter som kjører på tjener siden. En servlet instans er i stand til å motta HTTP anrop og generere et web-innhold (HTML) som svar. Det finnes ulike alternativer til hvordan servlets integreres med (eksisterende) web-tjenere:

- Hele web-tjeneren er implementert i Java (og kan lett utvides med servlets).
- Web-tjener og Java virtuell maskin (med servlets) kjører i separate prosesser. Disse bruker IPC (TCP sockets) og en egen protokoll for å kommunisere seg i mellom. Det er dette som er tilfelle med Apache Jserv (servlet støtte for den populære Apache web-tjener).
- Et tredje alternativ er å inkludere JVM i samme prosess som web-tjener. Apache prosjektet har også jobbet med dette. Det vil si de har vurdert et modul for apache som inkluderer JVM, men dette arbeidet ser ut til å ha stoppet opp.

3. Oppgaven

Vi skal designe og implementere et plugg-inn modul for Apache web-tjener, som kan utføre HTTP anrop ved hjelp av Java servlets . Dette modulet skal inkludere JVM i web-tjener prosessen og kalle Java metoder ved hjelp av Java Native Interface [1].

Vi skal gjøre en kvantitativ undersøkelse av ytelse (målinger) og en kvalitativ vurdering av fleksibilitet, robusthet og skalerbarhet i vår analyse av fordeler og ulemper med vårt design i forhold til den eksisterende Jserv som kjører JVM i separate prosesser. Kan vi fra våre resultater konkludere med at den ene arkitekturen er bedre enn den andre eller kommer det an på bruksmønster eller omgivelser?

4. "Open Source"

Apache er et eksempel på et såkalt "open source" prosjekt. Det er kanskje (ved siden av Linux) det mest vellykkede sådan, siden Apache nå er verdens mest brukte web-tjener. Vi skal egentlig forholde oss til to prosjekter under Apache paraplyen:

- Apache (web-tjener) prosjektet [2]. Apache er komponentorientert i den forstand at vi kan lage og plugge inn moduler som representerer utvidelser (for eksempel CGI, Perl scripts eller Java). Grensesnittet for slike moduler er godt dokumentert og det finnes verktøy for å kompilere slike moduler.

- Java Apache prosjektet [3] som jobber med servlets for Apache og tjenester som bygger på dette (implementert i Java).

Prosjektet skal søke utnytte det som allerede er gjort i Java-Apache prosjektet og gjerne gjenbruke kode derfra. Vi ønsker å gjenbruke design og kode fra Apache Jserv så langt som mulig. Det ligger derfor i kortene at koden vi utvikler også skal være "open source". Hvis vi kan bidra med kode og erfaringer tilbake til Java-Apache prosjektet ville det være bra!

5. Forkunnskaper

Kandidaten bør ha kjennskap til programmering i C og Java. Kandidaten bør ha kursene "operativsystem" og "klient-tjener programmering" eller tilsvarende. Videre er det en fordel å ha kurset "Åpne System" og det er en fordel med UNIX (Linux) erfaring (men ikke noe krav).

6. Veiledning og praktisk gjennomføring

Veiledere vil være Øyvind Hanssen, Jan P. Nytun og Mikael Snaprud (HiA og Agder IKT-senter).

Det er ønskelig at arbeidet gjøres med tilknytning til Agder IKT senter. Eksperimentet vil delvis måtte gjøres i en UNIX (Linux) omgivelse (Gullfisk).

Referanser

[1] Sheng Liang, "The Java Native Interface", Addison Wesley 1999, ISBN 0-201-32577-2

[2] Apache Project: http://www.apache.org

[3] Java Apache Project: http://java.apache.org

Hovedoppgavens tittel:

Effektiv integrasjon av Java servlets i Apache web-tjener

Student: Steffen S. Hellestøl

Veiledere: Øyvind Hanssen
Jan Nytun
Mikael Snaprud

Oppgaven går ut på å:

1. Gi en oversikt over forskjellige måter en Java servlet kan implementeres i web-tjeneren Apache. I første omgang ser det ut som at det finnes to måter; (i) den som brukes i dag, Apache JServog som kjøres i en egen prosess utenfor selve Apache serveren, og (ii) en løsning hvor JVM kjøres som en plugginn modul i samme prosess som selve Apache serveren. Jeg vil også undersøkes om det finnes andre realistiske arkitekturer utenom disse to.

2. Gjøre en sammenlikning av disse. Fordeler og ulemper ved hvert av løsningsalternativene må drøftes; for eksempel med hensyn til begrensninger, robusthet, sikkerhet, skalerbarhet og ytelse.

3. Implementere en Java servlet arkitekturen hvor JVM integreres i Apache, se punkt 1(ii).

4. Gjøre ytelsesmålinger for metode (i) og (ii) i forskjellige miljøer for å sammenligne de to arkitekturene. Med miljø menes for eksempel forskjellige hardware og software plattformer og forskjellige typer belastninger og bruksmønstre.

Jeg ønsker foreløpig å gjøre følgende avgrensning: Jeg vil kun fokusere på Linux implementasjonen av Apache. Jeg vil ikke ta hensyn til andre web-tjenere verken for Linux eller for andre operativsystem.

Referanse:

Forslag til diplomoppgave av Øyvind Hanssen, Agder IKT senter;
"Effektiv integrasjon av Java Servlets i web-tjener"
Publisert på http://fag.grm.hia.no/ikt6400/O99/servlet.diplom.pdf